In *International Journal on Artificial Intelligence Tools*, vol. **3**, No. 3, pp. 417-427, 1994.

LOCATION-INDEPENDENT TRANSFORMATIONS: A GENERAL STRATEGY FOR IMPLEMENTING NEURAL NETWORKS

George L. Rudolph and Tony R. Martinez
Computer Science Department
Brigham Young University Provo, Utah 84602
e-mail: george@axon.cs.byu.edu

Most Artificial Neural Networks (ANNs) have a fixed topology during learning, and typically suffer from a number of short-comings as a result. Variations of ANNs that use dynamic topologies have shown ability to overcome many of these problems. This paper introduces Location-Independent Transformations (LITs) as a general strategy for implementing neural networks that use static and dynamic topologies. A LIT creates a set of location-independent nodes, where each node computes its part of the network output independent of other nodes, using local information. This type of transformation allows efficient support for adding and deleting nodes dynamically during learning. Two simple networks, the single-layer competitive learning network, and the counterpropagation network, which combines elements of supervised learning with competitive learning, are used in this paper to illustrate the LIT strategy. These two networks are *localist* in the sense that ultimately one node is responsible for each output. LITs for other models are presented in other papers.

Keywords: Artificial Neural Networks, Hardware Implementation Design, Dynamic Topologies

1. Introduction

Artificial Neural Networks (ANNs) use a different computational paradigm than conventional von Neumann mechanisms. ANNs are composed of nodes and weighted connections between nodes, where each node computes its output based on a function of its weighted inputs. The overall function that a network computes is typically changed by altering the values of the weights between nodes, until the desired result is achieved. The main features of ANNs are learning ability, generalization, parallelism, self-organization and fault-tolerance. These features allow ANNs to solve various applications not handled well by current conventional computational mechanisms. Application areas include, but are not limited to, problems requiring learning, such as pattern recognition, control and decision systems, speech, and signal analysis [1].

Hardware support for ANNs is important for handling large, complex problems in real time. Learning times can exceed tolerable limits for complex applications with conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods.

Most ANNs use only static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies typically suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.
- local error minima during learning
- no *a priori* mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.)

Current research is demonstrating the use of dynamic topologies in overcoming these problems [2-5]. A *dynamic topology* is one which allows adding and deleting entire nodes and individual weighted connections during learning.

Early ANN hardware implementations are model-specific, and are intended to support only static topologies [6-8]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [9-11]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not Of course, general parallel machines, like the Connection Machine [12] and the CRAY [13], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

A LIT transformation redesigns the network so that each node contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed. These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of LI-nodes based on the original model.
2. Embed the nodes in a tree.

This process is outlined in figure 1. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right).

LIT is not simply a reorganization of the original network, but rather involves redesigning the basic structure. The heart of a transformation is the construction step: It defines the mapping between the original network and the set of LI-nodes. The mapping, in turn, affects how the behavior of the original is modeled in the transformed network. Thus, a construction typically also involves reformulating the network equations, in order to describe precisely the behavior of the transformed network. Although the original and transformed networks compute equivalent functions, correspondences between the two are not always direct, nor obvious. Constructions vary across different ANNs. LI-nodes for CL, which has only a single weight layer, do not have the same structure as LI-nodes for multilayer backpropagation.

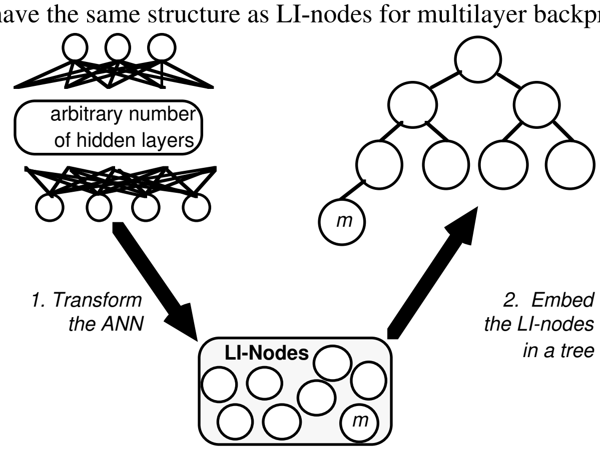

Figure 1. General LIT Transformation

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figure 1.. It is a simple, fast, regular topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used.

An example illustrates the support for dynamic topologies. When a node is added to (deleted from) an original network, both the node and its connections to other nodes are added (deleted). If a hidden node 6 were added to the original network in figure 2, four connections to the input nodes and three connections to the output nodes would also be added. This is accomplished in the transformed network by allocating a free node, such as the left child of node 3, and initializing it as a 4x3 node with the corresponding weights. The deletion of a hidden node in the original network, such as node 5, is accomplished in the transformed network by marking the corresponding node as "free". No other nodes are affected.

LITs have been developed for multilayer backpropagation [14], and Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [3], [15]. Transformations for other important ANNs are also being developed. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support inherently dynamic ANNs and dynamic variations of many static ANNs. A prototype VLSI chip has been designed and tested as proof-of-concept of the LIT strategy [16].

Two networks, the competitive learning (CL) network [17] with a single weight layer and the counterpropagation network (CPN) [18], are used in this paper to illustrate the LIT strategy. CL and CPN are examples of *localist* networks, i.e. for each possible output, there is a single node that is ultimately responsible for that output. CL uses unsupervised learning, while CPN combines elements of competitive and supervised learning.

Both the original CL and CPN models use static topologies, but this paper also outlines how LIT can support extended dynamic versions of these models. While support for dynamic versions of static ANNs is demonstrated, LIT is not concerned with devising such extensions, but rather seeks to support extensions being devised by others. LIT strategy assumes that dynamic extensions exist in the literature, as do inherently dynamic models such as RCE [5] and ASOCS [3]. Although the algorithms are very different, the transformation for RCE would be similar to that of CPN, and the transformation of competitive learning models using adaptive Radial-Basis Function Networks [19] (for example) would be similar to that of CL. In this paper, a simple dynamic extension to CL and CPN is used as a means of showing support for dynamic topologies.

The intent of this paper is to introduce the general LIT strategy—the descriptions given here should be seen as examples of transformations, not exclusive of other, more complex models. Furthermore, a complete transformation description requires formal definitions of the equations and algorithms of each model. For the purposes of this paper, and for brevity, only informal descriptions are given here, and it is assumed that the reader is familiar with the original models. More details on LITs for localist models can be found in [20].

Section 2 describes the LIT for CL. Section 3 describes the LIT for CPN. Section 4 is the conclusion.

2. Transformation Of A Competitive Learning Network

The goal of the CL model is to spontaneously classify sets of similar inputs to the same class, and sets of different inputs into different classes, according to critical features discovered by the network [17]. The original CL has one output node for each output class, and weights from each input node to every output node. The output nodes are classical sum-of-products nodes—the node with the highest activation for a particular input is chosen as the output class for that input. (The input nodes are place-holders for the input values.) Thus, the output of the network is the class of the input. During learning, only the winning node adjusts its input weights—the other nodes make no changes. A node causes its input weights to change so that the weights are more similar to the current input vector.

The original CL model reveals the localist nature of computation in the original CL model:

- The activation value for each class is computed locally at each output node.
- A single, global winner node determines the output of the network.
- Determination of a global winner can be accomplished using localized comparisons of subsets of the output nodes.

The LIT construction given here is based on these ideas.

Figure 2 shows the transformation of an 8x3 network into a LIT network with three 8x2 nodes. There is one node in the transformed network for each original output node. A transformed CL node stores two vectors. One vector contains weights on inputs $\boldsymbol{w_j}$, whose elements w_{ij} correspond to the weight connected to the respective original input node. The other vector contains the activation value net_j, and the class $class_j$ (or class number j) as a pair. The original node 2, in figure 2, has eight weights on inputs, and outputs its class number 2. The transformed node 2 has eight weights in $\boldsymbol{w_j}$ and outputs the tuple (net_j, $class_j$).

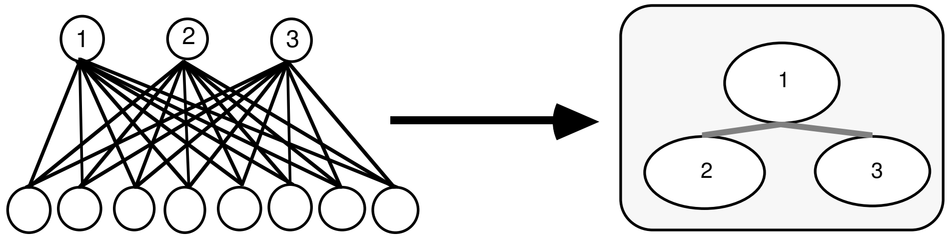

Figure 2. Transforming a CL Network

Figure 3 shows the structure of a new node in more detail. Each transformed node outputs its activation because determining a global winner requires non-local access to the activation values of each node. In the original network, the output nodes compete in some fashion, the node with the highest net activation being the winner, and all others being "quenched" somehow. The mechanisms for quenching are usually assumed, not explicitly given, but typically require that each node has access to the activation of every other node. The transformed network localizes this competition by means of a gather operation. Each node compares its own activation with the activations in the tuples received from its children. The tuple with the highest activation is sent to the parent, the others are discarded. In case of a tie, the tuple with the lowest class number is chosen. Thus, at any given transformed node, at most three tuples are compared. The output of the root node is the tuple which contains the class that has the highest activation value.

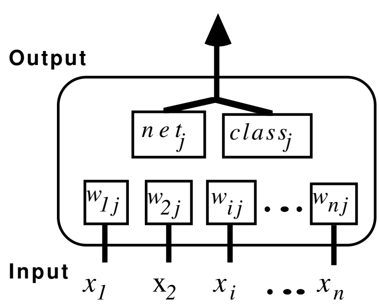

Figure 3. A CL Node

It may appear at first glance that the amount of parallelism in the network is degraded by the transformation—the transformed network has fewer nodes, and therefore each node does more work. This impression is false. The input nodes in an original CL network don't really *do* anything—conceptually, they are input buffers that hold the values of input variables. The real work is done at the output nodes (as there are no hidden nodes). The amount of real parallelism in computing the sum-of products is ultimately implementation-dependent, and LIT does not constrain implementation. Hence, the amount of parallelism is actually constrained by the number of output nodes, and by the physical implementation of a node. In any case, if there is not a large number of nodes, then there cannot be a lot of parallelism.

During execution, each node receives all the inputs by broadcast, and performs a *sum-of-products* function on the inputs and input weights. The result is each node's activation. Each node then sends its class and activation value up to its parent. Each parent compares its activation with those of its two children, and sends the class with the highest activation up to its parent. The others are ignored. The output at the root of the network (tree) is the output class to which the input vector belongs, along with the activation of the node corresponding to that class. The root node this single tuple to the Control Unit.

During learning, each node receives the input by broadcast, computes an activation, and sends its class and activation up to its parent (as above for execution mode). The Control Unit receives the class activation of the node with the highest activation. The output class is broadcast back to the network, in order to select the winning node, and "quench" all others. The selected node then alters its weights appropriately; no other nodes make changes.

The learning algorithm guarantees that only one node can win a competition. The learning equation shows that a winning node will change its input weights to respond more strongly to the current input.

The learning equation

$$Dw_{ij} = g(x_i / v - w_{ij})$$

for the standard CL model [17] gives a very non-robust type of learning. Both the zero vector and the vector with all ones will always be put in the class with the lowest number, because in each case, the activations of all nodes is the same. This helps to illustrate the generality of LIT—a different, more robust learning equation (algorithm) could be substituted in the original model, and the overall transformation would still be the same.

The original CL model above only supports a static topology—it describes how to change the weights, but provides no information about adding or deleting nodes and connections. An example illustrates the support for dynamic topologies. Assume that appropriate extended dynamic equations and algorithms exist: Typical schemes for adding nodes involve using a distance metric computed at each node. If the "distance" of the input pattern from the *winning* node is too far, a new output node is added to the network, and the original winning node is penalized in some fashion. A node can be deleted from the network if the node never wins a competition or if its average activation value is below some threshold over some period of learning. When a node is added to (deleted from) an original network, both the node and its connections to other nodes are added (deleted). If a new output node is added to the original network in figure 2, eight connections to the input nodes would also be added. Additional connections among the output nodes may be required also, depending upon how the global competition is handled. Adding a new output node is accomplished in the transformed network by allocating a free node, such as the left child of node 3, and initializing it as a 8x2 node with the corresponding weights and output values. The deletion of an output node in the original network, such as node 3, is accomplished in the transformed network by marking the corresponding node as "free". No other nodes are affected.

3. Transformation Of A Counterpropagation Network

The goal of CPN is to learn approximations to input-output pairs presented to the network [18]. The original network is presented as a four-layer "counterpropagation" flow between five sets of nodes (figure 4). However, , without loss of generality, CPN can be viewed as single-layer competitive learner with adjustable output weights.

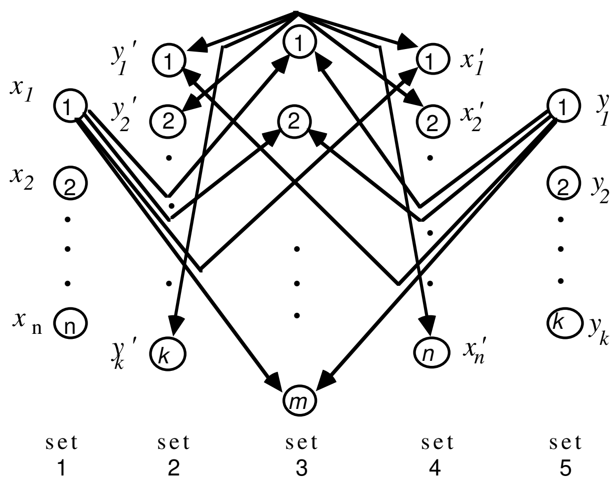

Figure 4. Original CPN Structure

Throughout section 3, it is necessary to refer to node layers and weight layers both. To avoid confusion, *layers* refers to weight layers, and *sets* refers to node layers.

The organization of the CPN can be simplified as follows:

1. Node sets 1 and 5 are combined into a single input set, divided into a $(\boldsymbol{x},\boldsymbol{y})$ vector pair as desired, where n is the size of $\boldsymbol{x}$, and k is the size of $\boldsymbol{y}$.
2. Node sets 2 and 4 are combined into a single output set, divided into a $(\boldsymbol{x}', \boldsymbol{y}')$ vector pair corresponding to $(\boldsymbol{x},\boldsymbol{y})$.
3. Node set 3 remains as in the original description.
4. There is a component-wise weight layer from sets 1 and 5 to sets 2 and 4 respectively, used only during learning. This is a pass-through layer whose weight values (set to 1) never change. After the transformation, it will not be needed.
5. The typical competitive method for determining the winner among the nodes of set 3 involves connections from each node in the set to every other node in the set. The transformed CPN network performs the selection in the same fashion as the CL network (see section 2). Hence, these connections can be ignored in the present discussion.

At this point, the network can be seen as a 2-layer (three, counting the pass-through layer) feed forward network with one set of hidden nodes (see figure 5), without loss of functionality. However, one more reorganizing step will show that CPN can be considered a competitive learner with the addition of adjustable weights as output. Several additional notational simplifications, along with relabelling (noted in figure 5), done for the purposes of this paper, are given to clarify this:

- The input vector pair $(\boldsymbol{x},\boldsymbol{y})$ is treated as a single vector $\boldsymbol{X}$, with components x_i, where $1 \le i \le k+n$.
- Similarly, the output vector pair $(\boldsymbol{x}', \boldsymbol{y}')$ is treated as a single vector $\boldsymbol{Y}$, with components y_i, where $1 \le i \le k+n$.
- The input weight vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are treated as a single weight vector $\boldsymbol{U}$, with components u_{ij}, where $1 \le i \le k+n$, $1 \le j \le m$
- Node set 3 is labeled $\boldsymbol{H}$, where h_i is the ith node in that set.
- The output weight vector $\boldsymbol{W}$ (for $\boldsymbol{x}'$ and $\boldsymbol{y}'$) is treated as a single vector $\boldsymbol{W}$, where w_{ij} is the weight from h_i to y_j, and $1 \le i \le m$, $1 \le j \le k+n$.

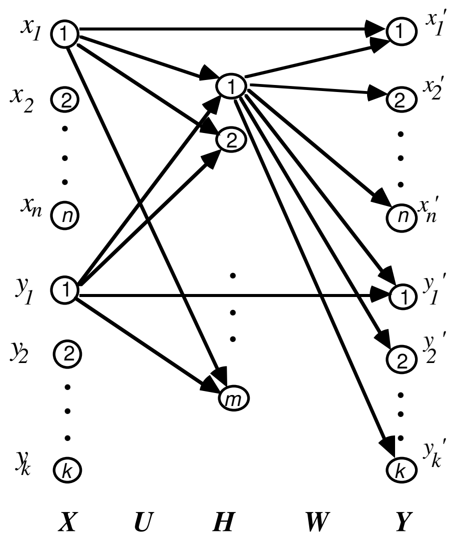

Figure 5. Rearranged CPN Network

The nodes are classical sum-of-products nodes. The nodes in set ***H*** compete in winner-take-all fashion, hence this hidden set acts as a selector for the output vector. Because ***H*** is a selector, there is no recombining of the weights in layer ***W*** to determine the output of the network, in either execution or learning mode. Only the winning node's output weights are (effectively) transmitted to the output layer ***Y***, because all other results are zero. This suggests that the real work of determining the output involves the weights in ***U***, the first weight layer, and the nodes in ***H***. Hence, CPN behaves as a spontaneous classifier like the original CL model of section 2: ***H*** corresponds to the output nodes, ***U*** corresponds to the single weight layer, and ***X*** corresponds to the input nodes. What CPN has, that CL does not, is ***W***, which are adjustable output values associated with each output class.

In transforming the CPN network, there are m nodes in the transformed network, one corresponding to each node in ***H*.** The structure of a transformed CPN node is shown in figure 6. Each node has a fixed class number, which is also its priority and index, from 1 to m. In case of a tie, the node with the lowest index wins. A node stores $2(k+n)$ weights: $k+n$ weights on inputs $\boldsymbol{u_j}$, and $k+n$ weights on output $\boldsymbol{w_j}$. The weights that each transformed node stores are the input and output weights of the corresponding original node.

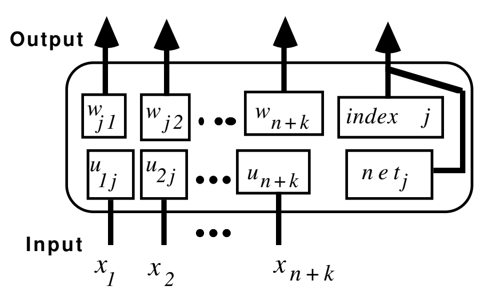

Figure 6. Structure of a Transformed CPN node

There are four ways in which the CPN network behaves differently from the CL network:

1. Each node outputs $\boldsymbol{w_j}$ in addition to net_j and $class_j$.
2. The winning node adjusts both its $\boldsymbol{w_j}$ and its $\boldsymbol{u_j}$.
3. The input and output vectors can contain real values.
4. The learning equations are different.

During execution, a node receives all the inputs by broadcast and performs a *sum-of-products* function on the inputs and input weights. The result is the node's activation. The node then sends its activation value and output class up to its parent.

In learning, only the winning node's weights are adjusted, just as in the standard CL model. Hecht-Nielsen's original equations allow for different learning constants for the ***x*** and ***y*** weights, but otherwise the equations for the two vectors are identical [18]. Each node receives the input, computes an activation (as above), and sends its activation and class up to its parent. The node with the highest activation is chosen, and its class is broadcast back to the network. The winning node then alters its weights on input *and* output appropriately. No other nodes make changes.

The original CPN model above only supports a static topology—it describes how to change the weights, but provides no information about adding or deleting nodes and connections. The idea used here to support dynamic topologies is essentially the same as for CL in section 3. Adding nodes involves using a distance metric computed at each node. If the "distance" of the input pattern from the *winning* node is too far, a new hidden node is added to the network, and the original winning node is penalized in some fashion. A hidden node can be deleted from the network if the node never wins a competition or if its average activation value is below some threshold over some period of learning.

Assume there is an original CPN network, like the one shown in figure 5, with $n=20$, $k=10$, and $m=10,000$. When a node is added to (deleted from) the original network, both the node and its connections to other nodes are added (deleted). If a new hidden node is added to the original network, 30 (i.e. $n+k$) connections to the input nodes and 30 connections to the output nodes would also be added. Additional connections among the hidden nodes may be required also, depending upon how the global competition is handled. In the transformed network, adding a new hidden node is accomplished by allocating a free node in the tree and initializing it as a 30x(30+2) node with the corresponding weights and index values. The deletion of a hidden node in the original network is accomplished in the transformed network by marking the corresponding node as "free". No other nodes are affected.

4. Conclusion

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings, such as parameter problems, local minima, and less than general optimization ability. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing static ANNs, static ANNs with dynamic extensions, and inherently dynamic ANNs.

This paper defined LITs for the competitive learning and counterpropagation networks. These ANNs are examples of the more general class of *localist* ANNs. The LIT descriptions for these two models have three main parts:

1. Construct a transformation (mapping) between the original ANN and the LIT uniform tree topology.
2. Demonstrate the equivalence of the original and transformed networks.
3. Since the original ANNs use static topologies, demonstrate how LIT supports extended versions which use dynamic topologies.

For ANNs that are inherently dynamic, and do not need to be extended, step 3 is subsumed by step 2.

Current work includes the following:

- transformations for neural networks other than those already developed,
- VLSI design and fabrication of LIT models.

Acknowledgments

This research is funded in part by grants from Novell Inc. The authors wish to express gratitude to Dr. Evangelos A. Yfantis for all his help, and to thank the reviewers and editors for accepting this paper.

References

[1] DARPA. Neural Network Study. AFCEA International Press, 1988.

[2] Fahlmann, Scott, C. Lebiere. *The Cascade-Correlation Learning Architecture.* in **Advances in Neural Information Processing 2.** Morgan Kaufmann Publishers: Los Altos, CA. pp. 524-532.

[3] Martinez, T.R., D.M. Campbell. *A Self-Adjusting Dynamic Logic Module*. **J. Parallel and Distributed Computing, Vol. 11, #4** (1991) 303-313.

[4] Odri, S.V., D.P. Petrovacki, G.A. Krstonosic. *Evolutional Development of a Multilevel Neural Network.* ***Neural Networks, Vol. 6, #4.*** Pergamon Press Ltd.: New York (1993) 583-595.

[5] Reilly, D.L., L.N. Cooper, C. Elbaum. *Learning Systems Based on Multiple Neural Networks.* (Internal paper). Nestor, Inc. (1988).

[6] Farhat, N., D. Psaltis, A. Prata, and E. Paek. *Optical Implementation of the Hopfield Model.* **Applied Optics, Vol. 24, #10.** (1985) 1469-1475 .

[7] Graf, H., L. Jackel, W. Hubbard. *VLSI Implementation of a Neural Network Model.* In **Artificial Neural Networks: Electronic Implementations,** Nelson Morgan, Ed. (1990) 34-42.

[8] Mead, Carver. *Analog VLSI and Neural Systems.* Addison-Wesley Publishing Company, Inc. (1989).

[9] Hammerstrom, D., W. Henry, M. Kuhn. *Neurocomputer System for Neural-Network Applications.* In **Parallel Digital Implementations of Neural Networks.** K. Przytula, V. Prasanna, Eds. Prentice-Hall, Inc. (1991).

[10] Ramacher, U., W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weißling, E. Schneider, R. Männer, J. Gläß. *Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1.* **Proceedings, World Congress on Neural Networks 1993, Vol. 4** (1993) 775-778.

[11] Shams, S. *Dream Machine—A Platform for Efficient Implementation of Neural Networks with Arbitrarily Complex Interconnect Structures.* **Technical Report CENG 92-23. PhD Dissertation,** USC. (1992).

[12] Hillis, W. Daniel. *The Connection Machine.* Cambridge, Mass.: MIT Press. (1985).

[13] Almasi, G., A. Gottlieb. *Highly Parallel Computing*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc. (1989).

[14] Rudolph G., Martinez, T. R. *An Efficient Transformation for Implementing Two-layer Feedforward Neural Networks.* To appear in the **J. Artificial Neural Networks** (1995).

[15] Rudolph G., and T.R. Martinez. *An Efficient Static Topology for Modeling ASOCS.* International Conference on Artificial Neural Networks, Helsinki, Finland. In **Artificial Neural Networks,** Kohonen et al, ed. North Holland: Elsevier Publishers. (1991) 729-734.

[16] Stout, M., G. Rudolph, T.R. Martinez, L. Salmon. *A VLSI Implementation of a Parallel, Self-Organizing Learning Model.* **Proceedings of the 12th IEEE International Conference on Pattern Recognition .**(1994) 373-376.

[17] Rumelhart, D., J. McClelland, et. al. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*. MIT Press (1986).

[18] Hecht-Nielsen, R. *Counterpropagation Networks.* **Applied Optics, Vol. 26, #23** (1987) 4979-4984.

[19] Haykin, Simon. *Neural Networks: A Comprehensive Foundation*. New York: Macmillan College Publishing Company, Inc. (1994) 266-268.

[20] Rudolph G., Martinez, T. R. *A Transformation for Implementing Localist Neural Networks.* To appear in **J. Neural, Parallel and Scientific Computations** (1995).